

NAME clearerr, feof, ferron, fileno – check and reset stream status

SYNOPSIS
#include <stdio.h>

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferron(FILE *stream);
int fileno(FILE *stream);
```

DESCRIPTION

The function **clearerr()** clears the end-of-file and error indicators for the stream pointed to by *stream*.
The function **feof()** tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function **clearerr()**.
The function **ferron()** tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the **clearerr()** function.
The function **fileno()** examines the argument *stream* and returns its integer descriptor.
For non-locking counterparts, see **unlocked_stdio(3)**.

ERRORS

These functions should not fail and do not set the external variable *errno*. (However, in case **fileno()** detects that its argument is not a valid stream, it must return **-1** and set *errno* to **EBADF**.)

CONFORMING TO

The functions **clearerr()**, **feof()**, and **ferron()** conform to C89 and C99.

SEE ALSO

open(2), **fdopen(3)**, **stdio(3)**, **unlocked_stdio(3)**

NAME fopen, fdopen, fileno – stream open functions

SYNOPSIS
#include <stdio.h>

```
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
int fileno(FILE *stream);
int fclose(FILE *stream);
```

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream with it.
The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences):
r Open text file for reading. The stream is positioned at the beginning of the file.
r+ Open for reading and writing. The stream is positioned at the beginning of the file.
w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** to a shared memory object is undefined.

The function **fileno()** examines the argument *stream* and returns its integer descriptor.

The **fclose()** function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush(3)**) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** is returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclose**, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

EBADF

The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for the routine **malloc(3)**.

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open(2)**.

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl(2)**.

malloc(3) malloc(3)

NAME calloc, malloc, free, realloc – Allocate and free dynamic memory

SYNOPSIS `#include <stdlib.h>`

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

`calloc()` allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

`malloc()` allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

`free()` frees the memory space pointed to by *ptr*, which must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behaviour occurs. If *ptr* is `NULL`, no operation is performed.

`realloc()` changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is `NULL`, the call is equivalent to `malloc(size)`; if *size* is equal to zero, the call is equivalent to `free(ptr)`. Unless *ptr* is `NULL`, it must have been returned by an earlier call to `malloc()`, `calloc()` or `realloc()`.

RETURN VALUE

For `calloc()` and `malloc()`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or `NULL` if the request fails.

`free()` returns no value.

`realloc()` returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or `NULL` if the request fails. If *size* was equal to 0, either `NULL` or a pointer suitable to be passed to `free()` is returned. If `realloc()` fails the original block is left untouched - it is not freed or moved.

CONFORMING TO ANSI-C

SEE ALSO `brk(2)`, `posix_memalign(3)`

getc/fgets/putc/fputs(3) getc/fgets/putc/fputs(3)

NAME `getc`, `fgets`, `getc`, `getchar`, `fputc`, `putc`, `fputs`, `putc`, `putchar` – input and output of characters and strings

SYNOPSIS `#include <stdio.h>`

```
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
int fputc(int c, FILE *stream);
int puts(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

DESCRIPTION

`fgetc()` reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or `EOF` on end of file or error.

`getc()` is equivalent to `fgetc()` except that it may be implemented as a macro which evaluates *stream* more than once.

`getchar()` is equivalent to `getc(stdin)`.

`fgets()` reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an `EOF` or a newline. If a newline is read, it is stored into the buffer. A `'\0'` is stored after the last character in the buffer.

`fputc()` writes the character *c*, cast to an *unsigned char*, to *stream*.

`fputs()` writes the string *s* to *stream*, without its terminating null byte (`'\0'`).

`putc()` is equivalent to `fputc()` except that it may be implemented as a macro which evaluates *stream* more than once.

`putchar(c)`; is equivalent to `putc(c, stdout)`.

Calls to the functions described here can be mixed with each other and with calls to other output functions from the *stdio* library for the same output stream.

RETURN VALUE

`fgetc()`, `getc()` and `getchar()` return the character read as an *unsigned char* cast to an *int* or `EOF` on end of file or error.

`fgets()` returns *s* on success, and `NULL` on error or when end of file occurs while no characters have been read. `fputc()`, `putc()` and `putchar()` return the character written as an *unsigned char* cast to an *int* or `EOF` on error.

`fputs()` returns a nonnegative number on success, or `EOF` on error.

SEE ALSO `read(2)`, `write(2)`, `ferror(3)`, `fgetc(3)`, `fgetwc(3)`, `fopen(3)`, `fread(3)`, `fseek(3)`, `fgetc(3)`, `getchar(3)`, `scanf(3)`, `ungetc(3)`, `write(2)`, `ferror(3)`, `fputc(3)`, `fopen(3)`, `fputwc(3)`, `fputws(3)`, `fseek(3)`, `fwrite(3)`, `gets(3)`, `putwchar(3)`, `scanf(3)`, `unlock_stdio(3)`

<p>opendir/readdir(3)</p> <p>NAME</p> <p>opendir — open a directory / readdir — read a directory</p> <p>SYNOPSIS</p> <pre>#include <sys/types.h> #include <dirent.h> DIR *opendir(const char *name); int closedir(DIR *dirp); struct dirent *readdir(DIR *dir);</pre> <p>DESCRIPTION</p> <p>The <code>opendir()</code> function opens a directory stream corresponding to the directory <i>name</i>, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.</p> <p>RETURN VALUE</p> <p>The <code>opendir()</code> function returns a pointer to the directory stream. On error, NULL is returned, and <i>errno</i> is set appropriately.</p> <p>DESCRIPTION</p> <p>The <code>closedir()</code> function closes the directory stream associated with <i>dirp</i>. A successful call to <code>closedir()</code> also closes the underlying file descriptor associated with <i>dirp</i>. The directory stream descriptor <i>dirp</i> is not available after this call.</p> <p>RETURN VALUE</p> <p>The <code>closedir()</code> function returns 0 on success. On error, -1 is returned, and <i>errno</i> is set appropriately.</p> <p>DESCRIPTION</p> <p>The <code>readdir()</code> function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by <i>dir</i>. It returns NULL on reaching the end-of-file or if an error occurred. It is safe to use <code>readdir()</code> inside threads if the pointers passed as <i>dir</i> are created by distinct calls to <code>opendir()</code>.</p> <p>The data returned by <code>readdir()</code> is overwritten by subsequent calls to <code>readdir()</code> for the same directory stream.</p> <p>The <i>dirent</i> structure is defined as follows:</p> <pre>struct dirent { long d_ino; /* inode number */ char d_name[256]; /* filename */ };</pre> <p>RETURN VALUE</p> <p>On success, <code>readdir()</code> returns a pointer to a <i>dirent</i> structure. (This structure may be statically allocated; do not attempt to <code>free(3)</code> it.)</p> <p>If the end of the directory stream is reached, NULL is returned and <i>errno</i> is not changed. If an error occurs, NULL is returned and <i>errno</i> is set appropriately. To distinguish end of stream and from an error, set <i>errno</i> to zero before calling <code>readdir()</code> and then check the value of <i>errno</i> if NULL is returned.</p> <p>ERRORS</p> <p>EACCES Permission denied.</p> <p>ENOENT Directory does not exist, or <i>name</i> is an empty string.</p> <p>ENOTDIR <i>name</i> is not a directory.</p> <p>GSP-Klausur Manual-Auszug 2023-02-22 1</p>	<p>stat(2)</p> <p>NAME</p> <p>stat, fstat, lstat — get file status</p> <p>SYNOPSIS</p> <pre>#include <sys/types.h> #include <sys/stat.h> #include <unistd.h> int stat(const char *path, struct stat *buf); int fstat(int fd, struct stat *buf); int lstat(const char *path, struct stat *buf);</pre> <p>Feature Test Macro Requirements for glibc (see <code>feature_test_macros(7)</code>):</p> <pre>lstat(0) _BSD_SOURCE _XOPEN_SOURCE >= 500</pre> <p>DESCRIPTION</p> <p>These functions return information about a file. No permissions are required on the file itself, but — in the case of <code>stat()</code> and <code>lstat()</code> — execute (search) permission is required on all of the directories in <i>path</i> that lead to the file.</p> <p><code>stat()</code> stats the file pointed to by <i>path</i> and fills in <i>buf</i>.</p> <p><code>lstat()</code> is identical to <code>stat()</code>, except that if <i>path</i> is a symbolic link, then the link itself is stat-ed, not the file that it refers to.</p> <p><code>fstat()</code> is identical to <code>stat()</code>, except that the file to be stat-ed is specified by the file descriptor <i>fd</i>.</p> <p>All of these system calls return a <i>stat</i> structure, which contains the following fields:</p> <pre>struct stat { dev_t st_dev; /* ID of device containing file */ ino_t st_ino; /* inode number */ mode_t st_mode; /* protection */ nlink_t st_nlink; /* number of hard links */ uid_t st_uid; /* user ID of owner */ gid_t st_gid; /* group ID of owner */ dev_t st_rdev; /* device ID (if special file) */ off_t st_size; /* total size, in bytes */ blksize_t st_blksize; /* blocksize for file system I/O */ blkcnt_t st_blocks; /* number of blocks allocated */ time_t st_atime; /* time of last access */ time_t st_mtime; /* time of last modification */ time_t st_ctime; /* time of last status change */ };</pre> <p>The <i>st_dev</i> field describes the device on which this file resides.</p> <p>The <i>st_rdev</i> field describes the device that this file (inode) represents.</p> <p>The <i>st_size</i> field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.</p> <p>The <i>st_blocks</i> field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than <i>st_size/512</i> when the file has holes.)</p> <p>The <i>st_blksize</i> field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)</p> <p>GSP-Klausur Manual-Auszug 2023-02-22 1</p>	<p>opendir/readdir(3)</p> <p>NAME</p> <p>opendir — open a directory / readdir — read a directory</p> <p>SYNOPSIS</p> <pre>#include <sys/types.h> #include <dirent.h> DIR *opendir(const char *name); int closedir(DIR *dirp); struct dirent *readdir(DIR *dir);</pre> <p>DESCRIPTION</p> <p>The <code>opendir()</code> function opens a directory stream corresponding to the directory <i>name</i>, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.</p> <p>RETURN VALUE</p> <p>The <code>opendir()</code> function returns a pointer to the directory stream. On error, NULL is returned, and <i>errno</i> is set appropriately.</p> <p>DESCRIPTION</p> <p>The <code>closedir()</code> function closes the directory stream associated with <i>dirp</i>. A successful call to <code>closedir()</code> also closes the underlying file descriptor associated with <i>dirp</i>. The directory stream descriptor <i>dirp</i> is not available after this call.</p> <p>RETURN VALUE</p> <p>The <code>closedir()</code> function returns 0 on success. On error, -1 is returned, and <i>errno</i> is set appropriately.</p> <p>DESCRIPTION</p> <p>The <code>readdir()</code> function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by <i>dir</i>. It returns NULL on reaching the end-of-file or if an error occurred. It is safe to use <code>readdir()</code> inside threads if the pointers passed as <i>dir</i> are created by distinct calls to <code>opendir()</code>.</p> <p>The data returned by <code>readdir()</code> is overwritten by subsequent calls to <code>readdir()</code> for the same directory stream.</p> <p>The <i>dirent</i> structure is defined as follows:</p> <pre>struct dirent { long d_ino; /* inode number */ char d_name[256]; /* filename */ };</pre> <p>RETURN VALUE</p> <p>On success, <code>readdir()</code> returns a pointer to a <i>dirent</i> structure. (This structure may be statically allocated; do not attempt to <code>free(3)</code> it.)</p> <p>If the end of the directory stream is reached, NULL is returned and <i>errno</i> is not changed. If an error occurs, NULL is returned and <i>errno</i> is set appropriately. To distinguish end of stream and from an error, set <i>errno</i> to zero before calling <code>readdir()</code> and then check the value of <i>errno</i> if NULL is returned.</p> <p>ERRORS</p> <p>EACCES Permission denied.</p> <p>ENOENT Directory does not exist, or <i>name</i> is an empty string.</p> <p>ENOTDIR <i>name</i> is not a directory.</p> <p>GSP-Klausur Manual-Auszug 2023-02-22 1</p>	<p>stat(2)</p> <p>NAME</p> <p>stat, fstat, lstat — get file status</p> <p>SYNOPSIS</p> <pre>#include <sys/types.h> #include <sys/stat.h> #include <unistd.h> int stat(const char *path, struct stat *buf); int fstat(int fd, struct stat *buf); int lstat(const char *path, struct stat *buf);</pre> <p>Feature Test Macro Requirements for glibc (see <code>feature_test_macros(7)</code>):</p> <pre>lstat(0) _BSD_SOURCE _XOPEN_SOURCE >= 500</pre> <p>DESCRIPTION</p> <p>These functions return information about a file. No permissions are required on the file itself, but — in the case of <code>stat()</code> and <code>lstat()</code> — execute (search) permission is required on all of the directories in <i>path</i> that lead to the file.</p> <p><code>stat()</code> stats the file pointed to by <i>path</i> and fills in <i>buf</i>.</p> <p><code>lstat()</code> is identical to <code>stat()</code>, except that if <i>path</i> is a symbolic link, then the link itself is stat-ed, not the file that it refers to.</p> <p><code>fstat()</code> is identical to <code>stat()</code>, except that the file to be stat-ed is specified by the file descriptor <i>fd</i>.</p> <p>All of these system calls return a <i>stat</i> structure, which contains the following fields:</p> <pre>struct stat { dev_t st_dev; /* ID of device containing file */ ino_t st_ino; /* inode number */ mode_t st_mode; /* protection */ nlink_t st_nlink; /* number of hard links */ uid_t st_uid; /* user ID of owner */ gid_t st_gid; /* group ID of owner */ dev_t st_rdev; /* device ID (if special file) */ off_t st_size; /* total size, in bytes */ blksize_t st_blksize; /* blocksize for file system I/O */ blkcnt_t st_blocks; /* number of blocks allocated */ time_t st_atime; /* time of last access */ time_t st_mtime; /* time of last modification */ time_t st_ctime; /* time of last status change */ };</pre> <p>The <i>st_dev</i> field describes the device on which this file resides.</p> <p>The <i>st_rdev</i> field describes the device that this file (inode) represents.</p> <p>The <i>st_size</i> field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.</p> <p>The <i>st_blocks</i> field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than <i>st_size/512</i> when the file has holes.)</p> <p>The <i>st_blksize</i> field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)</p> <p>GSP-Klausur Manual-Auszug 2023-02-22 1</p>
--	--	--	--

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See "noatime" in **mount(8)**.)

The field *st_atime* is changed by file accesses, for example, by **execve(2)**, **mknod(2)**, **pipe(2)**, **utime(2)** and **read(2)** (of more than zero bytes). Other routines, like **mmap(2)**, may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod(2)**, **truncate(2)**, **utime(2)** and **write(2)** (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following POSIX macros are defined to check the file type using the *st_mode* field:

- S_ISREG(m)** is it a regular file?
- S_ISDIR(m)** directory?
- S_ISCHR(m)** character device?
- S_ISBLK(m)** block device?
- S_ISFIFO(m)** FIFO (named pipe)?
- S_ISLNK(m)** symbolic link? (Not in POSIX.1-1996.)
- S_ISSOCK(m)** socket? (Not in POSIX.1-1996.)

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

ERRORS

EACCES Search permission is denied for one of the directories in the path prefix of *path*. (See also **path_resolution(7)**.)

EBADF

fd is bad.

EFAULT

Bad address.

ELOOP

Too many symbolic links encountered while traversing the path.

ENAMETOOLONG

File name too long.

ENOENT

A component of the path *path* does not exist, or the path is an empty string.

ENOMEM

Out of memory (i.e., kernel memory).

ENOTDIR

A component of the path is not a directory.

SEE ALSO

access(2), **chmod(2)**, **chown(2)**, **fstat(2)**, **readlink(2)**, **utime(2)**, **capabilities(7)**, **symlink(7)**

NAME

strtok, **strtok_r** – extract tokens from strings

SYNOPSIS

#include <string.h>

char *strtok(char *str, const char *delim);

char *strtok_r(char *str, const char *delim, char **saveptr);

DESCRIPTION

The **strtok()** function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok()** the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

Each call to **strtok()** returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, **strtok()** returns NULL.

A sequence of calls to **strtok()** that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. The first call to **strtok()** sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in *str*. If such a byte is found, it is taken as the start of the next token. If no such byte is found, then there are no more tokens, and **strtok()** returns NULL. (A string that is empty or that contains only delimiters will thus cause **strtok()** to return NULL on the first call.)

The end of each token is found by scanning forward until either the next delimiter byte is found or until the terminating null byte (0) is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and **strtok()** saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, **strtok()** returns a pointer to the start of the found token.

From the above description, it follows that a sequence of two or more contiguous delimiter bytes in the parsed string is considered to be a single delimiter, and that delimiter bytes at the start or end of the string are ignored. Put another way: the tokens returned by **strtok()** are always nonempty strings. Thus, for example, given the string "aaa:bbb", successive calls to **strtok()** that specify the delimiter string ":", would return the strings "aaa" and "bbb", and then a null pointer.

The **strtok_r()** function is a reentrant version **strtok()**. The *saveptr* argument is a pointer to a *char ** variable that is used internally by **strtok_r()** in order to maintain context between successive calls that parse the same string. On the first call to **strtok_r()**, *str* should point to the string to be parsed, and the value of *saveptr* is ignored. In subsequent calls, *str* should be NULL, and *saveptr* should be unchanged since the previous call.

Different strings may be parsed concurrently using sequences of calls to **strtok_r()** that specify different *saveptr* arguments.

RETURN VALUE

strtok() and **strtok_r()** return a pointer to the next token, or NULL if there are no more tokens.

ATTRIBUTES

Multi-threading (see pthreads(7))

The **strtok()** function is not thread-safe, the **strtok_r()** function is thread-safe.